Exploiting Replication*

Kenneth P. Birman
Thomas A. Joseph

TR 88-917
June 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037. Contract N00140-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp Date Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) TR 88-917	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Kenneth P. Birman, Assist. Prof. CS Dept., Cornell University 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Defense Advanced Research Porject Agency/ISTO
6c ADDRESS (City, State, and ZIP Code)	7b ADDRESS (City, State, and ZIP Code) Defense Advanced Research, Project Agency Attn: TIO/Admin., 1400 Wilson Blvd. Arlington, VA 22209-2308
8a NAME OF FUNDING / SPONSORING ORGANIZATION DARPA/ISTO 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code) See 7b.	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Exploiting Replication

12 PERSONAL AUTHOR(S)
Kenneth P. birman & Thomas A. Joseph

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical (Special	FROM _____ TO _____	June 1988	51

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This is a preprint of material that will appear in the collected lecture notes from Arctic 88, An Advanced Course on Operating Systems, Trõmso, Norway, July 5-14, 1988. The lecture notes will appear in book form later this year.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Exploiting Replication*

Kenneth P. Birman
Thomas A. Joseph

Department of Computer Science
Cornell University
Ithaca, NY 14853

June 1, 1988

This is a preprint of material that will appear in the collected lecture notes from *Arctic 88, An Advanced Course on Operating Systems*, Trömso, Norway, July 5-14, 1988. The lecture notes will appear in book form later this year.

* This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N00140-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Arctic 88 Course Outline

1. Introduction
   1.1. Evolution of Distributed Systems
   1.2. Five perspectives on Distributed Systems
   1.3. Architecture
   1.4. Distribution transparency
   1.5. An architecture for selecting transparency
   1.6. Modelling systems

2. Technology
   2.1. Hardware Substrates and relevant software
   2.2. Networks

3. Communication
   3.1. Interprocess Communication
   3.2. Reliable Broadcast protocols
   3.3. Remote Procedure Call

4. Naming and Protection
   4.1. Naming
   4.2. Cryptography
   4.3. Protection

5. Concurrency and Consistency,
   5.1. Transactions
   5.2. Using transactions in distributed applications
   5.3. Theory of (nested) transactions
   5.4. Using replication to enhance availability and fault-tolerance in distributed systems
   5.5. Virtual synchrony for building distributed systems

6. File Systems
   6.1. Basics
   6.2. Why distributed file systems?
   6.3. Brief history
   6.4. Clarification of nomenclature
   6.5. Andrew File System
   6.6. Context
   6.7. Design
   6.8. Future

7. System Management
   7.1. Management Architectures
   7.2. Debugging, Reconfiguration control
   7.3. Accounting
   7.4. Effects of scale

8. Formal methods
   8.1. The tools of formal logic
   8.2. Representing behaviour in LOTOS
   8.3. High-level specifications for distributed programs

9. Conclusions

Chapter 1

Exploiting replication

1.1 Using replication to enhance availability and fault-tolerance in distributed systems

The focus of this chapter is on the use of data replication and replicated execution to obtain faster response time or fault-tolerance in distributed programs. These techniques can be critical in determining whether or not a network-based solution to an application problem will be feasible.

For example, modular expansion and *price-performance* considerations argue for the use of distributed systems in factory automation settings. However, many factories contain devices controlled by dedicated processors that require realtime response. Any delay imposed on the controllers by the network must be bounded. This is hard to ensure because of possible packet loss and unpredictable load on remote servers. Consequently, such systems are forced to replicate or cache data needed by the controllers.

Similarly, distributed execution may be valuable in a factory automation setting, because it enables applications to take advantage of the enormous computational resources available in a network. To distribute the execution of a request over multiple processes, however, it is necessary to replicate the data structures used to coordinate their actions. Otherwise, one cannot ensure that the processes will give consistent behavior. Thus, we need to understand replication to employ this type of distributed computation.

Fault-tolerance considerations can also motivate the use of replicated data and execution. In a non-distributed setting a failure rarely affects anything but the user of the crashed program or machine. In a network, the effects of a crash can ripple through large numbers of machines. A program

that will survive the failures of programs with which it interacts must have access to redundant copies of critical resources and ensure that its state is never dependent, even indirectly, on information to which only the failed program had access. It may also be necessary to maintain backup processes that will take over from a failed process and complete time-critical computations or computations that have acquired mutual exclusion on shared resources.

This chapter explores a number of approaches to replication and distributed consistency issues. The treatment is applicable to a conventional local area network or a loosely coupled multiprocessor. The programs and computers in such systems fail benignly, by crashing without sending out incorrect messages. Processors do not have synchronized clocks, hence the failure of an entire site can thus only be detected *unreliably*, using timeouts. Message communication is assumed to be reliable but bursty, because packets can be lost and may have to be retransmitted.

Two major problems that arise in these settings will not be considered here. The first is network partitioning, where the network splits into subnetworks between which communication is impaired (for example, if a LAN bridge fails). Providing replication that spans partitions is a difficult problem and an active research area. Secondly, we will not look at problems that place realtime constraints on distributed algorithms or protocols. Realtime issues are hard to isolate; once they are introduced, the entire system must often be treated from a realtime perspective. Although our methods are potentially useful in systems for which a realtime constraint leads the designer to dedicate a computer to some device, we will assume that the realtime aspects of such problems do not extend beyond the control program itself.

1.2 The tradeoff between shared memory and message passing

At the heart of any distributed system lies the problem of transferring information between cooperating processes. Broadly speaking, this can be done in one of two ways: by permitting the processes to interact with some common but passive resource or memory, or by supporting message exchange between them. There are advantages and disadvantages associated with each approach, hence the most appropriate style of information transfer for a particular problem must be determined by an analysis of the characteristics of that problem. In database systems, the shared memory paradigm is

the [illegible]e, and many other systems share this character. In other settings, however, a shared resource might represent a bottleneck that could be avoided using replication and direct message-based interactions between the processes using that resource.

This point is important because the approach used to replicate data depends strongly on way in which processes will interact. Database replication techniques, such as were described in Chapter ??, are quite different from replication for processes that interact directly via message passing.

The focus of this chapter will be on message oriented systems. We start by identifying a set of characteristics of problems that call for message-based solutions, as opposed to ones based on shared memory. This characterization leads to a list of services that a system must address in order to permit message-based solutions to distributed applications using replicated data. Next, we look at a number of systems in order to understand how they address the problems in this list. Finally, we examine a particular model for solving these problems in a message-passing environment and a set of solutions that can be easily understood in terms of this model.

1.3 Consistent distributed behavior in distributed systems

Shared memory has been studied intensively since distributed computing systems were first proposed. The dominant problem that must be addressed here is to ensure that the processes using a shared memory be able to coordinate their access to it so as to achieve "consistent" behavior with respect to one another (a concept that we will look at in more detail below). Transactional serializability, as discussed in previous chapters, is a widely accepted solution to this problem. This leads to a natural question: should *all* types of distributed consistency be viewed as variant forms of transactional consistency, or are there problems that can only be addressed using other methods? Looking at the factory automation setting, it is not hard to find problems that fall outside the traditional transactional framework. Consider the following two examples:

> • ***Build software for monitoring job status and materials inventories. Updates will be done by the warehouses (quantities on hand), "cell controllers" (requests for materials and changes in job status), and from a central management site (changes in***

prices, deliveries from suppliers, changes in job priorities, etc). Queries will be done from managerial officies throughout the factory complex.

- *Develop software for a cell controller operating a set of drills. Each drill is independently controlled by a dedicated microprocessor. The cell as a whole receives a piece of work to do, together with a list of locations, sizes and tolerances for the holes to be drilled. It must efficiently schedule this work among the drills. Drills can go offline for maintenance or because of bits breaking, or come online while the cell is active, hence the scheduling problem is dynamic. Some drills are better suited to heavy low-precision work, while others are suitable for lighter high-precision work. Finally, it is critical that a hole not be drilled twice, even if a drill bit breaks before it is fully drilled, because this would result in a very low precision. Instead, an accurate list of partially drilled holes should be produced for a human technician to check and redrill manually.*

These two problems illustrate very different styles of distributed computing, and consequently distributed consistency means something different for each. The former clearly lends itself to a transactional approach. One would configure the various programs into a "star", with a database at the center, perhaps replicated for fault-tolerance. Programs throughout the network interact through the database. Transactions are the natural consistency model for this setting. The essential observation to make is that the processes share data but are *independent*. By adopting a transactional style of interaction, they can avoid tripping over one another. Moreover, transactions provide a simple way to ensure that even if failures occur, the database remains intact and consistent.

Now, consider the second problem. A star configuration seems much less natural here. The processes in this example need explicit knowledge of one another in order to coordinate their actions on a step-by-step basis. They need to reconfigure in response to events that can occur unpredictably, and to ensure the consistency of their views of the system state and one-another's individual states. When a control process comes online after being offline for a period of time, it will have to be reintergrated into the system, in a consistent way which may have very little to do with its state at the time of the failure. On the other hand, when a process goes offline, the processes

that remain online need to assume responsibility for finishing any incomplete work and generating the list of holes to be manually checked. Moreover, it is not reasonable to talk about "aborting" partially completed work, since this could result in redrilling a hole.

What should consistency mean in this case? All of the above considerations run contrary to the spirit of a transactional approach, where the goal is serializability – *non* interference between processes. A process in a transactional system is encouraged to run as if in isolation, whereas the cell controller involves explicit interactions and interdependencies between processes. Transactions use aborts and rollback to recover from possibly inconsistent states, but in this example rollback is physically impossible. On the other hand, although the kind of consistency here may not be transactional, one would not want to go to the extreme of concluding that there is no meaningful form of consistency that applies in this setting. Certainly, there should be a reasonable "explanation" for what each control process is doing, and this explanation should be in accordance with the cell controller specification.

This leaves us with two choices. One option is to look at how the transactional model could be extended to cover these new requirements. This approach has lead to mechanisms like *top-level* transactions[1] [Liskov83], mixtures of serializable and non-serializable behaviors [Lynch86,Herlihy86a], and specialized algorithms for concurrently accessing data structures like B-tree indexes. The trouble is that these introduce complexity into a model that was appealing for its simplicity. Moreover, these methods have been around for some time, and have proved appropriate only for a narrow set of problems. The second option, pursued here, is to develop a different style of distributed computation better matched to problems like the ones arising in a cell controller.

[1] Top-level transactions are a striking example of how the tradeoff between shared memory and message passing can lead to complex system structures. A top-level transaction is essentially a way of sending a message from "within" the scope of an uncommitted transaction to other transactions running outside that scope. It provides an escape from the shared memory paradigm into the message passing one. The fact that such a mechanism is needed within transactional systems is strong evidence that no single approach addresses all types of distributed system.

1.4 Tools for building directly distributed software

For lack of a better term, let us refer to problems in this class as being *directly distributed,* since they involve processes that interact directly with one another rather than indirectly through a shared resource. The remainder of this chapter examines operating system tools for solving directly distributed problems. Directly distributed applications arise in many settings other than the factory floor. Consequently, these sorts of tools would apply to any collection of processes that need to closely coordinate their actions and share data, or tolerate failures.

Transactional "tools" provide for synchronization, data management, transaction commit, and so forth. Tools for a directly distributed system would provide analogous mechanisms oriented towards helping the members of some group of cooperating processes behave in a mutually consistent way. Typically, this would include some subset of the following:

Process groups: A way to form an association between a set of processes for cooperating to solve a problem.

Group communication: A location-transparent way to communicate with the members of a group or a list of groups and processes. In some systems, group communication consists only of a way to find some single member of the named group. In others, communication is broadcast-oriented [2] and *atomic*, meaning that all members of the destination group receive a given message unless a failure occurs, in which case either all the survivors receive it or none does. A problem that must be addressed is how group communication should work when the group membership is changing at the time the communication takes place. Should the broadcast be done before the change, after it, or is it acceptable for some group members to observe one ordering and some the other? Should message delivery to an unresponsive destination be retried indefinitely, or eventually interrupted – with the attendant risk that the destination was just experiencing a transient failure and is actually still operational? We will see that the way in which a system

[2] A group broadcast should not be confused with a hardware broadcast. A group broadcast provides a way to communicate with all members of some group. It might or might not make use of hardware facilities for broadcasting to all the machines connected to a local area network. Here, unless we explicitly indicate that we are talking about a hardware broadcast, the term broadcast will always mean broadcast to a group.

resolves these issues can determine the type of problems that process groups in the system can be used to solve.

Replicated data: A mechanism permitting group members to maintain replicated data. Most approaches provide a 1-copy consistency property, analogous to 1-copy serializability.

Synchronization: Facilities for synchronization of concurrent activities that interact through shared data or resources.

Distributed execution: Facilities for partitioning the work required to solve a problem among the members of a process group.

State monitoring mechanisms: Mechanisms for monitoring the state of the system and the membership of process groups, permitting processes to react to the failure of other group members.

Reconfiguration mechanisms: Facilities with which the system can adapt dynamically to failures, recoveries, and load changes that impact on work processing strategies.

Recovery mechanisms: Mechanisms for automating recovery, which could range from a way to to restart services when a site reboots to facilities for reintegrating a component into an operational system that is actively engaged in distributed computations.

More will be needed than a set of tools if the intention is to solve real-world distributed computing problems. Questions of methodology, efficiency of the implementation, and scalability must also be addressed. For example, it is easy to solve database problems using transactions. To be able to say the same about directly distributed software, one would need to demonstrate that the tools lead to a natural and intuitive programming style in which problems can be isolated and solved one by one, in a step-wise fashion. And, it must be easy to establish that the solutions will tolerate the concurrency and configuration changes characterizing asynchronous distributed systems.

One problem is that we lack a rigorous statement about what *consistent behavior* means in directly distributed settings. In a shared memory setting, consistent behavior generally means that the accesses made to the data by client programs are serializable [Bernstein83], and that some invariant holds on the state of programs themselves. In an message-oriented distributed setting, we don't have a data manager or shared data items, hence the

serializability constraint is lost. Instead, one would like to say that the processes in the system, taken as a group, satisfy some set of system-wide invariants in addition to local ones on their states.

Unfortunately, we are looking at *asynchronous* systems. When one says that two actions taken at different locations are in accord with a global predicate, that statement will have no meaning until it is decided *when* the predicate should be evaluated. This temporal dependency is particularly striking if the notion of consistency changes while the system executes. For example, consistent behavior in an idle cell controller is quite different from consistent behavior while work is present. Taking a more extreme example, consistent behavior of a distributed program for controlling a nuclear reaction means one thing during normal operation, but something entirely different if a cooling pump malfunctions. Since the switch from one rule to another cannot occur instantaneously, defining consistency rigorously is a hard problem.

Distributed systems designers have approached the consistency issue in several ways. Much theoretical work starts with a rigorous notion of distributed consistency. However, this work often relies on simplified system models that may not correspond to real networks. For example, the theoretical study of Byzantine agreement establishes limits on the achievable behavior of a distributed agreement protocol. The failure modes permitted include malicious behaviors that real systems do not experience, and the model assumes that all processors share a common clock (so that they can run in lock-step). The cost of Byzantine agreement turns out to be too high for any "real" system to pay. Similarly, innumerable papers have presented complex protocols to solve distributed problems, remarkably few of which have ever been implemented. Any practitioner who scans the literature discovers that many of these are in fact not "implementable" because they make unrealistic assumptions.

At the other extreme, most existing "distributed" operating systems provide little more than a message-passing mechanism, often available only through a cumbersome and inflexible communication subsystem. Systems like this simply abandon any rigorous form of consistency in favor of probabilistic behavioral statements. When attempts have been made to formally specify the behavior of real distributed systems, the results have often included so much detail that it becomes hard to separate the abstract behavior of the system from the implementation and interface it provides. Thus, a formal specification of a distributed system often includes details of how the message channels work, how addressing is handled, and so forth. While this

information is of value in designing applications that depend on a precise characterization of system behavior, high level issues such as "consistency" are obscured by such a treatment. As we will see, few of the problems in our list could be solved using a message-passing approach, and a highly detailed formalism describing exactly how the message-passing mechanism works offers little help.

An intermediate approach, which will be adopted here, restricts system behavior in order to simplify the solutions to problems like the ones that arise in the toolkit. On the one hand, these restrictions must be efficiently implementable. On the other, it must be possible to talk in abstract terms about how distributed programs execute in the system, what it means for them to behave consistently, and how consistency can be achieved. Our discussion will revolve around a model and associated orthogonality properties:

Model: Given a distributed system, we would like to be able to describe its behavior formally in a way that will help establish the correctness of algorithms run under it. If this requires restrictions on the permissible behavior of the system, we will need to understand how those restrictions can be enforced and how weak they can be made.

Orthogonality: Once one begins to make formal statements about the behavior of a set of tools for building directly distributed software, it becomes reasonable to ask questions about the extent to which the tools influence each other. Ideally, one would want tools that operate completely independently from one another. Otherwise, by extending the functionality of a system in one way, one would risk breaking the preexisting code.

Efficiency is also an important consideration. Nobody will use a set of tools unless it yields programs that perform as well (or better) than software built using other methods. Moreover, the absolute level of performance achieved must be good enough to support the kinds of applications likely to employ direct distribution.

A final issue relates to questions of scale. Our tools focus on direct distribution as a problem "in the small". One also needs to compose larger systems out of components built using these tools, in a way that isolates the larger-system issues from the implementation of the directly-distributed components of which it is built. Otherwise, it may be impractical to talk about system design and interface issues without simultaneously addressing implementation details.

1.5 System support for direct interactions between processes

A variety of systems provide some level of support for direct distribution. Let us look at how these address the major items in our list of tools.

1.5.1 Basic RPC mechanisms

Most operating systems provide *remote procedure calls* [Birrell84]. The technological support for remote procedure calls has advanced rapidly during the past decade, and sub-millisecond RPC times for inter-site communication should be common in operating systems in the near future. RPC does not, however, address any of the problems in the above list. Thus, the programmer confronted by a direct distribution problem would be in a very difficult situation when using a system for which RPC is the primary communication mechanism. Short of building a complex application-level mechanism to resolve these problems, there would seem to be no way to build directly distributed software.

1.5.2 Quorum methods

Many systems manage replicated data using quorum schemes (see Chapter ??). In these, update operations require two phases, while read-only operations can be done in one phase. An update is transmitted during the first phase, and then committed in the second phase if a quorum of copies were updated and aborted otherwise. The abort is needed to avoid an uncertain outcome if some processes failed just after doing the update but before getting a chance to reply. Consequently, recovering processes must start by determining the status of uncommitted operations that were underway at the time of the failure.

Although read operations can be done in one phase, they need to access a sufficient number of copies to ensure intersection with the writes. In a fault-tolerant setting, this number must exceed the number of simultaneous failures that the system is expected to tolerate.[3]

[3]Unless failed processes never recover, any fault-tolerant quorum scheme that allows writes during failures will use a write-quorum size smaller than the total number of processes with copies of the replicated data item. Since read quorums must always intersect write quorums, the read quorum size must be larger than the number of simultaneous failures that the system is designed to tolerate.

Thus, fault-tolerant quorum-based schemes require a complex recovery mechanism, and execute both reads and updates synchronously.

Do quorum methods offer solutions for the more general set of direct distribution mechanisms enumerated above? For example, consider the problem of synchronizing the actions of a set of processes. One might do this using a token managed with rules like the following:

> *A set of processes shares exactly one copy of a toker, using operations to pass and request it. If the holder of the token fails, a pass is done automatically on its behalf. New processes can join the set dynamically.*

Notice first that this problem is hard to solve using remote procedure calls. Typical RPC implementations detect failures using timeouts. Since timeouts can be inaccurate, an agreement protocol is needed to deal with token holder failures. The reader may want to try and design such a protocol: it isn't easy! For example, one could try to inform all operational processes of each pass, so that they know which process to request the token from. However, in addition to the inaccuracy of the failure detection mechanism, the solution must deal with the possibility that the token could be in motion at the time of a request. Dynamic group membership changes would make these problems even more complicated.

It would certainly be possible to use quorum methods to update variables identifying the current token holder and request queue. However, a fault-tolerant quorum-based token passing algorithm will be costly both in terms of code required and the performance that it can achieve. Thus, on the one hand it is hard to see how a non-quorum scheme could be used to solve this problem, but on the other, it is hard to believe that a quorum solution could perform at an acceptable level.

Token passing is just a simple example of the sort of problems that a directly distributed system would have to solve. In a setting where token passing is difficult, such systems will surely be impractical to construct. Some experimental evidence to support this claim exists: many systems support RPC's and quorum replication methods, but few provide mechanisms like the token passing facility outlined above. For example, the DEC VAX-Clusters system has a synchronization facility similar to this token mechanism [Kronenberg86]. However, the implementation is complex, and few application designers could undertake a similar effort.

1.5.3 The ARGUS system

The ARGUS language extends an RPC mechanism with transactional features and persistent data [Liskov83]. ARGUS provides support for recovery from failure, but not process groups or replicated data. Moreover, ARGUS uses an abort mechanism that rolls back actions in the event of a failure. This makes it easy to implement quorum replication methods, and consequently ARGUS would be a good environment for implementing a quorum-based token passing mechanism. On the other hand, transactions and rollback could lead to problems in settings like the factory automation example we looked at above.

1.5.4 The CAMELOT system

Like ARGUS, CAMELOT is a transaction-based system [Spector88]. The objective of the project is to achieve the highest possible transactional performance. Quorum replication techniques have been pursued vigorously in CAMELOT as part of the AVALON language built on top of the system [Herlihy88]. AVALON focuses on mechanisms for maintaining quorum-based replicated objects in which quorum sizes change dynamically [Herlihy86b]. This approach works well in settings with frequent communication partitioning, site failures and recoveries. The disadvantage is that quorum schemes are highly synchronous. We will be looking at systems that can solve the toolkit problems using methods that are *asynchronous* except when a site failure or network partitioning occurs. Such an approach will outperform a quorum scheme provided that these types of failures are rare, which is the case in most local area networks.

1.5.5 The V system

V is an RPC-based system that places a strong emphasis on performance and on system support for forming process groups and broadcasting requests [Cheriton85]. Recall that this raises the problem of how the group broadcast mechanism should work when group membership is changing or processes fail. Although V makes a "best effort" to deliver messages to all members of a process group, V gives no absolute guarantees that all receive a given broadcast, or that messages are received in some consistent order relative to a membership change.

A V-style broadcast is well suited for some types of directly distributed applications. If an application is broadcasting to a network resource man-

ager, for example to find the mailbox for a user, it may not matter very much if some processes fail to receive the request. It is easy to program around the uncertainty, ensuring that behavior is correct in all but the most improbable scenarios. Thus, when broadcasting mailbox location updates, it may not matter if some processors miss occasional updates [Lampson86]. In this example, and in others with a similar character, the V broadcast primitive is suitable for implementing replication. On the other hand, our token passing problem is no easier to solve in V than in a standard RPC setting. Similarly, replicated data with a 1-copy behavior constraint is hard to provide using the standard V broadcast: if an update fails to get through, or two updates arrive in different orders at different group members, the copies could end up with inconsistent values. To resolve this, a non-trivial mechanism would be needed at the application level.

1.5.6 The Linda S/Net

AT&T has developed an operating system, the Linda S/Net [Carriero86], using a different approach to the problem of direct distribution. Linda is a backplane interconnect that provides a reliable high speed hardware broadcast facility. Every processor on the bus can broadcast requests that all other processors will see. Bus contention logic ensures that messages are delivered without loss in the same order everywhere. The S/Net uses this to support a form of shared memory based on the idea of a shared collection of *tuples*. The operations provided are *out* (add a tuple to the space), *in* (read and remove a tuple) and *read* (read a tuple without deleting it). A pattern matching mechanism is used so that tuples can be extracted selectively by specifying the values of some fields and just the data types for others. All processors can resolve each *in* or *out* request in the same way by knowing only the order in which it was read off the network. Token passing would be a simple problem in Linda, as is the management of replicated data: the whole tuple space is fully replicated. Linda has been used primarily for building parallel software for problems ranging from numerical matrix computations to artificial intelligence. Nearly full utilization of the processors is often cited. On the other hand, the overhead of the full replication scheme can become significant in some algorithms.

For example, Figures 1.1 and 1.2 illustrate a skeletal solution to the drilling problem, using Linda tuples to describe the work to be done and the outcome. Each hole to be drilled is described by a "pending work" tuple. A drill control processor selects a tuple on which to work, drills the

```
NewBatch(hole_list)
  {
    /* Load workspace with hole descriptions */
    for(each hole in hole_list)
        out("to_do", hole.x, hole.y, ...);

    /* Wait for all to be processed */
    for(each hole in hole_list)
    {
        /* in will block until outcome is known */
        in("done", hole.x, hole.y, int status);
        if(status == MUST_CHECK)
            print("Must check hole at ...\n", hole.x, hole.y);
    }
  }
```

Figure 1.1: Generating work for a cell controller in the Linda-S/Net

hole, and then outputs a tuple describing the outcome.

Linda lacks a failure detection mechanism, so it is not clear how to extend this solution to generate a list of holes that a technician should recheck if a control process can crash without generating a MUST_CHECK tuple first. Likewise, it is hard to see how one could handle dynamic scheduling using Linda's tuple-matching mechanism.

Despite these limitations, the Linda system points to a possible solution for the kind of problems we are interested in. The essential observation is that when all processes in a system execute the same actions in the same order, distributed consistency is easily achieved. Unfortunately, the Linda hardware does not scale to large networks. Moreover, the basic Linda approach does not scale either: in a network with hundreds or thousands of nodes, a scheme in which every processor must receive every request would simply turn the network into a terrible bottleneck. Thus, it is best to view the Linda S/Net as more of an interesting case study than as a potential solution to directly distributed programming in a large loosely coupled network.

```
/* A typical drill controller */
DrillControl()
  {
    forever
    {
        in("to_do", int x, int y, ...);

        /* Position the drill, then make the hole */
        PositionDrill(x, y);
        outcome = DrillHole(HoleSpecs);

        /* Record outcome */
        out("done", x, y, outcome);
    }
  }
```

Figure 1.2: A control process for a single drill

1.5.7 The HAS system

At IBM, the HAS project explored a related approach to achieving globally synchronous broadcasts. HAS focused on support for Δ-*common storage*, which is much like the Linda tuple space but defined in terms of abstract operations on a shared memory. Updates are completed within a period of time bracketed by upper and lower bounds. Unlike Linda, HAS was designed for loosely coupled processors communicating on a high speed token ring. HAS was therefore built using a collection of message-passing protocols that achieve a strongly ordered atomic delivery property starting from unreliable message passing between processors with unsynchronized internal clocks [Cristian86,Cristian88]. The HAS methodology provides tolerance to a wide range of process and communication failure modes, even including Byzantine failure modes in which processes can behave in arbitrary malicious ways and experience bizarre clock failures. Unfortunately, the performance of HAS was poor because the *lower* bound on update times turns out to be a surprisingly large number regardless of system load – a substantial fraction of a second even for a small network of fast machines. Few applications can tolerate delays this large, and the project was ultimately discontinued.

1.5.8 The ISIS system

Like Linda and HAS, the ISIS system adopts an approach based on synchronous execution, whereby every process sees the same events in the same order. However, ISIS seeks good performance in larger local area networks. ISIS starts with the observation that synchronous execution models offer strong advantages. Their primary disadvantage is one of cost: without hardware support, a distributed lock-step execution performs poorly. Even with hardware support, a lock-step style of computation does not scale.

To address this, ISIS provides an *illusion* of synchronous execution, in much the same sense that transactional serializability provides the illusion of a sequential transaction execution. Whenever possible, ISIS relaxes synchronization in order to reduce the degree to which processes can delay one another and to better exploit the parallelism of a distributed environment. For example, processes are permitted to initiate an operation asynchronously, by broadcasting a request without pausing to wait for a reply[4]. On the other hand, the system will behave as if such messages were delivered immediately. Similarly, ISIS delivers broadcasts with common destinations in the same order everywhere – except when it is possible to infer that the application does not need such strong ordering. In such cases, ISIS can be told to relax the delivery ordering rules, which permits it to use a cheaper broadcast protocol.

ISIS differs from Linda and HAS in providing a message-oriented rather than a shared memory interface. The basic ISIS facilities include tools for creating and managing process groups, group broadcast, failure detection and recovery, distributed execution and synchronization, etc. A Linda-style replicated tuple space is easy to implement in ISIS, as is 1-copy replicated data. Moreover, the implementations can readily be customized to address special requirements of the application program, such as the selection of

[4]Note the difference between this and an RPC, where such a pause is built in even if no reply is desired. For example, on the SUN 3 version of ISIS a program that issues an asynchronous broadcast to 5 destinations would resume executing after a delay lasting for a small fraction of a millisecond. The remote message deliveries occur within 5-10 milliseconds. With RPC, which has a 10 millisecond round-trip time under UNIX on a SUN 3, the caller would be delayed by 50 milliseconds, plus any costs associated with the group addressing protocol. Delivery would take as long as 45 milliseconds between the start of the broadcast and the arrival of the last message. On systems with faster processors and cheaper RPC costs, the costs here might scale, but the same argument could still be made. The point here is that ISIS is sending acknowledgement messages, just as for the RPC, but concurrently with the continued execution by the sender.

the optimal next hole for a controller to drill. We will be looking at these mechanisms in more detail below, and then at an example illustrating how ISIS could be used to solve the drill control problem.

1.6 An execution model for virtual synchrony

One desirable feature of systems like ARGUS, CAMELOT, Linda, HAS and ISIS is that one can write down a model describing the execution environment they provide. In the case of ARGUS or CAMELOT the model is based on nested transactions, and the lowest-level elements are data items and operations upon them. Models for the latter systems are similar but oriented towards the representation of synchronous executions. Before looking at virtually synchronous algorithms for the tools enumerated earlier, it will be helpful to start by defining such a model and giving virtual synchrony a more precise meaning.

The elements of the execution model we will be working with are processes, process groups, and broadcast events. Broadcast events include more than group communication. Point-to-point messages are treated as a broadcast to a singleton process-group. Failures are treated as a kind of broadcast too: a last message from the dying process informing any interested parties of its demise. Data items are not explicitly represented, although one can superimpose a higher level on top of this basic model in which operations and the values of data become explicit.

1.6.1 Modeling a synchronous execution

One way to understand a model is as a formalism for writing down what an "external observer" might see when watching the system execute from someplace outside of it. The external observer provides a notion of global time to relate the actions taken by distinct processes. One defines an execution to be *synchronous* if the external observer can confirm that whenever two processes observe the same event, they do so at the same instant in time. This is illustrated in Figure 1.3, where time advances from top to bottom.

In a synchronous model it is easy to specify the meaning of an *atomic* ("all or nothing") broadcast to a process group. At the time at which a broadcast is delivered, it must be delivered to all *current* members of the group. Thus the set of destinations is determined by the event sequence (processes joining or leaving the group) that occurred prior to that time. This does not tell us how to implement such a broadcast, but it does give a

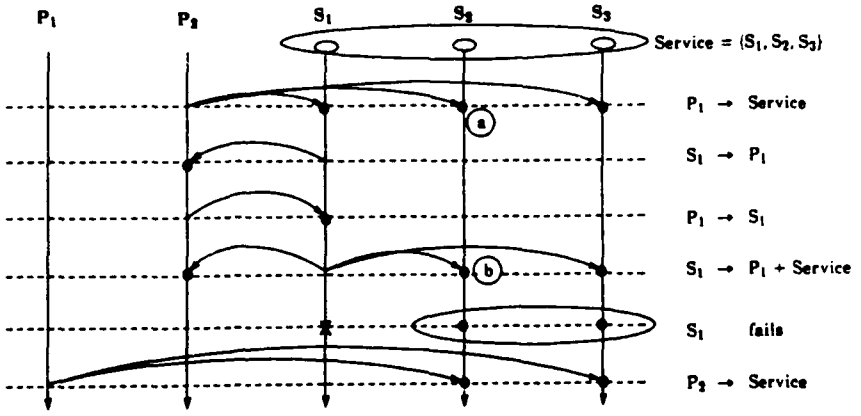

Figure 1.3: A synchronous execution. A pair of client processes, identified as P_1 and P_2 interact with a process group containing three server processes. Execution advances from top to bottom in a lock-step manner. Several message exchanges and a failure are shown.

rule for deciding whether a broadcast is a atomic or not. We will be making use of this rule below.

Among the above systems, Linda comes closest to providing a synchronous execution. However, a genuinely synchronous execution would be impractical to implement in a local area networking environment. To do so all the processes would need access to a common clock and to execute at fixed speeds, neither of which is normally possible.

1.6.2 Modeling a loosely synchronous execution

An execution is said to be *loosely synchronous* if all processes observe events in the *same order*. Figure 1.4 illustrates such an execution. An external observer who notes the time at which events are executed may see the same event processed at different times by different processes. However, the events will still be executed in the same order as they would have been in a truly synchronous execution. Hence if the system is not a realtime one (and this is something we assumed at the outset), processes that behaved correctly in a truly synchronous setting should still behave correctly in a loosely synchronous one [Neiger87].

More formally, for every loosely synchronous execution E, there exists

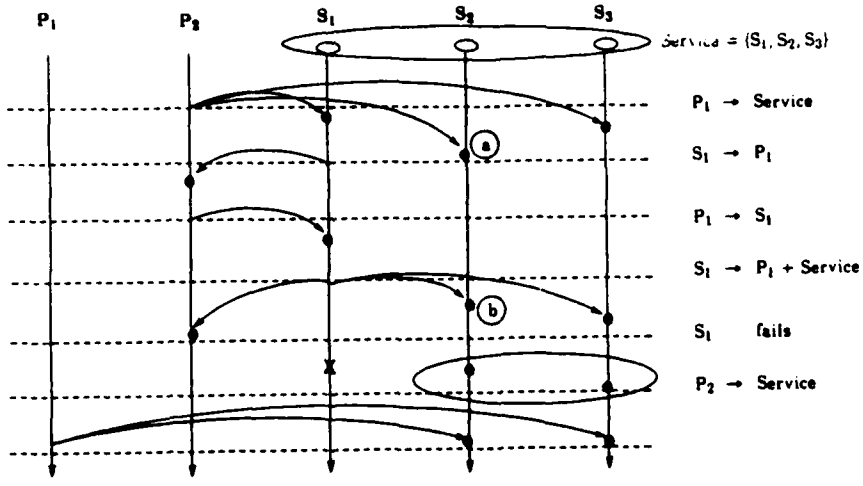

Figure 1.4: A loosely synchronous execution

an *equivalent* truly synchronous execution E'. The two executions are equivalent in the following sense. Let E_p be the sequence of events observed by process p in an execution E. Then $E'_p = E_p$ for all p, i.e. every process observes the same sequence of events in E and E'. Unless a process has access to a realtime clock, it cannot distinguish between E and E'. Figure ?? is indistinguishable from Figure 1.3 under this definition.

Any synchronous system is also loosely synchronous. Thus, Linda and HAS are both loosely synchronous systems; the global event ordering being imposed by hardware in the former case, and by a software protocol in the latter.

1.6.3 Modeling a virtually synchronous execution

A *virtually synchronous* execution is related to a loosely synchronous one in much the same way that a *serializable* execution is related to a serial one. The characteristic of a virtually synchronous system is that although an external observer may see cases in which events occur in different orders at different processes, the processes themselves are unable to detect this. For example, Figure 1.5 is a copy of Figure 1.4 with the delivery of event a delayed to occur after b at one destination. We would call this execution virtually synchronous if, after both a and b have terminated, no process in the system can contradict a claim that a executed first everywhere. Evidence of the order in which operations took place could be explicit in the value of

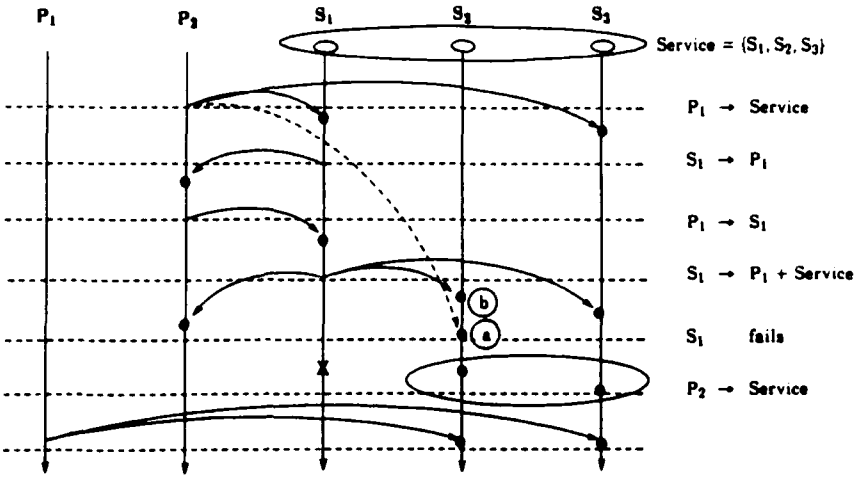

Figure 1.5: A virtually synchronous execution

some variable, or it could be reflected in the response to one of the requests or the actions that a process took after receiving some request.

Recalling the definition of loose synchrony in the previous section, an execution E is *virtually synchronous* if there exists some truly synchronous execution E' equivalent to E. However, we broaden our notion of equivalence between executions by requiring only that $E'_p \approx E_p$, for all p. Here "$\approx$" means that two event sequences are *indistinguishable*, but not necessarily identical. The determination of which event sequences are distinguishable depends on the semantics of the individual events in a particular application. A formal definition of this sort of equivalence and a theory of virtual synchony have been developed by Schmuck [Schmuck87].

Compared to loose synchrony, virtual synchrony puts less restrictions on system executions. As we will see below, this can be exploited to obtain more efficient implementations for many distributed systems.

1.7 Comparing virtual synchrony with other models

1.7.1 Transactional serializability

Is our model really any different than a transactional one? We argue that virtual synchrony is a substantial generalization of transactional serializabil-

ity.

Clearly, if a system is serializable, it is virtually synchronous. On the other hand, a virtually synchronous execution need not be serializable. First, there is nothing like a *transaction* in a virtually synchronous system. Consider a pair of processes, executing concurrently, that interchange a series of messages leading to a dependency of each on the state of the other. In a transactional setting, this could only occur if each interaction was a separate top-level transaction – a series of atomic actions with no subsuming transactions at all. However, transactional work has generally not considered this case directly, and it is normally not even stated that the serialization order for such top-level actions should be the order in which they were initiated. For many concurrency control schemes, such as two-phase locking, there is no *a-priori* reason that this would be the case: a single transaction might asynchronously initiate two top-level transactions, first T_1 and then T_2, which would be serialized in the order T_2 followed by T_1.[5]

Virtual synchrony imposes an explicit correctness constraint on sequences of interactions like this, namely that unless the order is irrelevant, the events be observed in the order they were initiated, even if they were initiated asynchronously, and even if order arises through a very indirect dependency of one action on another. Moreover, virtual synchrony talks about process groups and distributed events (broadcasts, failures, group membership changes, ...), whereas transactions only address replicated data. In light of the importance of these mechanisms for the directly distributed tools we listed earlier, and the apparent difficulty of layering them on top of transactions, these are significant differences.

Transactions and virtual synchrony both depend strongly on the semantics of operations. In the case of transactions, this was first observed when an attempt was made to extend transactional serializability to cover abstract data types [Schwartz84]. Whereas it is easy to talk about concurrency control and serializability for transactions that read and write (possibly replicated) data items, it is much harder to obtain good solutions to these problems for transactions on abstract data types. In the case of virtual synchrony, the problem arises because the model lacks data items or any other fixed referent with well-known semantics. One can only decide if an execution is virtually synchronous if one knows a great deal about how the system executes. This is an advantage in that the definition is consider-

[5] For example, T_1 might block waiting for a lock and then update variable x, while T_2 acquires its locks and manages to update x before T_1.

ably more powerful than any data-oriented one. There are many virtually synchronous systems that could not be interpreted as synchronous by somehow making the model knowledgeable about data. On the other hand, the presense of semantic knowledge makes it hard to talk about correct or efficient system behavior in general terms, without knowing what the system is doing. As we will see shortly, one can only do this through a detailed analysis of those algorithms on which a particular system relies.

1.7.2 Virtual synchrony in quorum-based schemes

Earlier, we saw some examples of how a quorum scheme might be used to obtain consistent behavior in a relatively unstructured setting. Such an approach can be understood as a form of virtual synchrony. The basic characteristic of a quorum scheme is its *quorum intersection relation*, which specifies how large the quorums for each type of operation must be [Herlihy86b]. If two operations potentially conflict – that is, if the outcome of one could be influenced by the outcome of the other – then their quorums will intersect at one or more processes. One can thus build a partial order on operations, such that all conflicting operations are totally ordered relative to one another, while non-conflicting operations are unordered. Since non-conflicting operations always commute, the executions of a quorum-based system are indistinguishable from any extension of this order into a total one. Such a total order can be understood as a description of a synchronous execution that would have left the system in the same state as it was in after the quorum execution. Thus, a quorum execution is virtually synchronous.

1.8 System support for virtual synchrony

1.8.1 The ISIS virtually synchronous toolkit

Let us now return to the ISIS system and look more carefully at some of the virtually synchronous algorithms on which it is based.

1.8.2 Groups and group communication

The lowest level of ISIS provides process groups and three broadcast primitives for group communication, called CBCAST, ABCAST and GBCAST. The primitives were discussed previously, in Chapter ??, and their integration into a common framework supporting group addressing is covered

elsewhere [Birman87a]. We therefore focus on their joint behavior while omitting implementation details.

In ISIS, a *process group* is an association between a group address and some set of members. Membership in a process group has low overhead, hence it is assumed that processes join and leave groups casually and that one process may be a member of several groups.

A *view of a process group* is a list of its members, ordered by the amount of time they have belonged to the group. ISIS includes tools for determining the current view of a process group and for being notified of each view change that occurs. All members see the same sequence of views and changes.

The destination of a broadcast in ISIS is specified as a list of groups. Group membership changes are synchronized with communication, so that a given broadcast will be delivered to the members of a group in the same membership view.

Recall that a broadcast is *atomic* if it is delivered to *all* members of each destination group. Here, "all" refers to all the group members listed in the process group view in which delivery takes place, which might not be the same as the membership when the broadcast was initiated.[6] A *virtually atomic* delivery is one in which all group members *that stay operational* receive the message in the same view. The ISIS broadcast primitives are all virtually atomic. Thus, the recipient of an ISIS broadcast can look at the "current" group membership (in a virtually synchronous sense) and act on the assumption that all of the listed processes also received the message. It may subsequently see some of them fail, perhaps without having acted on the message.

CBCAST, ABCAST and GBCAST differ in their delivery ordering properties. Before we review these differences, recall the definition of the potential causality relation on events, $\rightarrow$, introduced in Chapter ??: $e \rightarrow e'$ means that there may have been a flow of information from event e to event e' along a chain of local actions linked by message passing.

Let *BCAST(a)* denote the initiation of broadcast a and *DELIVER(a)* the delivery of some a to some destination. All three flavors of broadcast ensure that if *BCAST(a)* $\rightarrow$ *BCAST(b)* for broadcasts a and b (abreviated $a \rightarrow b$), then *DELIVER(a)* will precede *DELIVER(b)* at any common destinations. In fact, they satisfy an even stronger property, namely that if $a \rightarrow b$ then even if a and b have no common destinations, b will be delivered only if a can be delivered too. This ensures that if some subsequent broadcast c

[6] In ISIS, it will be the same or a subset of the initial membership.

is done, with $b \rightarrow c$, and a and c have common destinations, the system will be able to respect its delivery order constraints. The ISIS delivery ordering constraint can be thought of as a FIFO rule based not on the order in which individual processes transmitted broadcasts, but on the order in which threads of control did so. Here, a thread of control is any path along which execution may have proceeded.

CBCAST satisfies exactly the above delivery constraint. If a and b are concurrent, then CBCAST might deliver a and b in different orders. ABCAST provides a delivery order that extends $\rightarrow$ so that if a and b are two concurrent ABCAST's, a delivery order will be picked and respected at all shared destinations. However, ABCAST and CBCAST are unordered with respect to each other. GBCAST, in contrast, provides totally ordered delivery with respect to *all* sorts of broadcasts. Thus, if g is a GBCAST and a is any sort of broadcast then g and a will be delivered in a fixed relative order to all shared destinations.

A system that uses only ABCAST to transmit broadcasts is loosely synchronous. For this reason, ISIS uses ABCAST as its default protocol when not told otherwise by the programmer. However, ABCAST is costly. Like the quorum protocols, it sometimes delays message deliveries in a way that would be noticeable to the sender. CBCAST is much cheaper, especially when invoked asynchronously.[7] This leads to the question of just when synchronization can be relaxed in a broadcast algorithm.

1.8.3 When can synchronization be relaxed?

Let us examine the degree to which some specific algorithms depend on the ordering characteristics of the broadcasts used for message transmission. We begin with some examples drawn from a single process group with fixed membership:

- A shared tuple space, supporting the Linda *in*, *out* and *read* operations.
- A shared token, supporting operations to *request* it, to *pass* it, and to determine the current holder.

[7] The implementation of ISIS is more complex than the earlier discussion of these protocols made it appear. For reasons of brevity, the associated issues are not discussed here. However, the reader should be aware that to make effective use of protocols such as these, a substantial engineering investment is needed. This ranges from the requirement for a system architecture that imposes low overhead to heuristics for scaling the protocols to run in large networks and to avoid thrashing when communication patterns overload the most costly aspects of the protocols [Birman88].

- Replicated data. There are two cases: a variable that can be updated and accessed at will, and a variable that can only be accessed by the holder of a token (lock) on it. We look only at the second case, as the first one is essentially the same as for the Linda tuple space.

1.8.4 Shared tuple space

The Linda tuple space requires a very strong, globally observed ordering on operations. Except for the *read* operation, done locally without issuing a broadcast, all operations change the tuple space. Consequently, they all potentially conflict with one another. Of course, it is possible to relax the relative ordering when two operations affect independent tuples. However, it is unlikely that a system could take advantage of this because it would be necessary to look at the actual arguments of the operations, and not just their types.

1.8.5 Shared token

The shared token is interesting because it admits a variety of possible implementations. The most synchronous implementation is the easiest to understand. In this algorithm, both *request* and *pass* operations are transmitted using a globally ordered group broadcast. Members maintain a queue of pending requests. A token holder wishing to do a pass operation first waits until at least one request is pending, then broadcasts the pass operation. On receiving such a broadcast, all processes mark the request at the head of the queue as having been granted.

What if we wanted to use a cheaper broadcast primitive? Since the algorithm depends on a totally ordered request queue, we cannot use a cheaper protocol for sending requests without major algorithmic changes. On the other hand, it might be possible to use a less ordered protocol for transmitting *pass* operations. This, however, raises a subtle issue. It may be possible for a *request* message to reach one group member much earlier (in realtime) than some other. If we change the broadcast primitive, such a sequence could result in a race, where a *pass* operation arrives at a slow process before the request from which it will be satisfied (Figure 1.6). Likewise, a process about to pass the token could have received a request that has already been granted, but not have seen the pass message. Clearly, this would lead to error.

Fortunately, although the situations described above could arise when

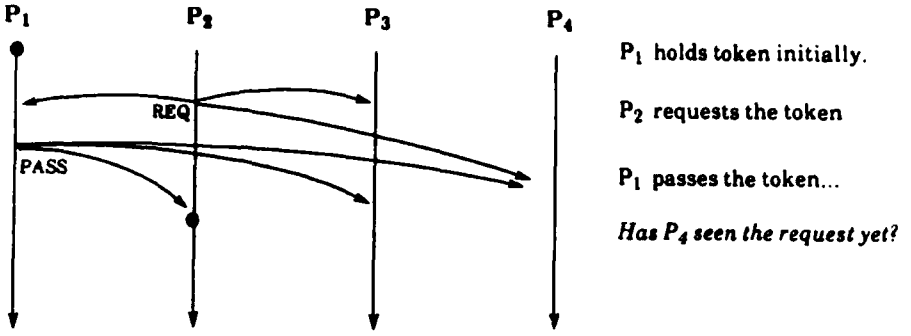

Figure 1.6: A race could develop when using a weakly ordered broadcast

using a totally unordered protocol, or one that is FIFO on a point-to-point basis, it cannot occur with a CBCAST protocol. To see this, notice that a process cannot try to pass the token unless it has first requested it and then received it from some other holder. Let R_i denote the *i'th* token request to be satisfied and P_i the pass done by the process that issued R_i. We have $R_i \rightarrow P_i$ and $\forall j < i : P_j \rightarrow P_i$. Thus, $\forall j < i : R_j \rightarrow P_i$. In other words, when CBCAST delivers a particular *pass* message, the destination will always have received the prior *request* operations and *vice versa*, eliminating the source of our concern.

This reasoning inspires a further refinement. Why not transmit *request* operations using CBCAST as well? The preceding analysis shows that any process receiving a *pass* will have received the *request* to which that pass corresponds. Thus, the only problem this change would introduce is due to the loss of a global request ordering: different processes could now receive requests in different orders. This means that it will no longer be possible for each process to determine, in parallel with the others, the new holder of the token: they would have no basis for making consistent decisions. On the other hand, the decision could be made by the process about to send a *pass* message. If there is no pending request, that process will have to defer its *pass* until a request turns up. Given a *pass* message that indicates the identity of the new holder, all processes can find and remove the corresponding request from their set of pending requests, where it will necessarily be

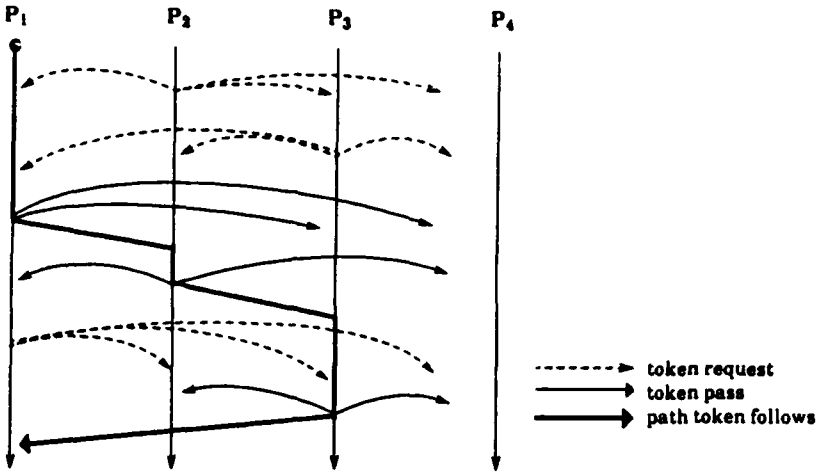

Figure 1.7: A virtually synchronous token-passing algorithm

found.

Figure 1.7 illustrates this behavior schematically. The darker lines show the path along with the token is passed, which is precisely the part of the $\rightarrow$ relation used in the above argument.

Thus, a token-passing problem admits a variety of correct solutions. The cheapest of these, from the point of view of message transmission, is the third. It depends only on the ordered delivery of messages that relate to one another under $\rightarrow$. A slight price is paid when a *pass* operation is done and there is no pending *request*: the broadcast corresponding to the *pass* must be delayed, and this will have the effect of introducing a delay before the *request* can be satisfied when it is finally issued. In the ISIS system, the benefits of using an asynchronous one-phase protocol to implement the broadcast outweigh any delay incurred in this manner.

Token passing is an especially interesting problem because it captures the essential behavior of any system with a single locus of control that moves about the system, but remains unique. Many algorithms and applications have such a structure. Thus, if we can solve token passing efficiently, there is some hope for solving a much larger class of problems efficiently as well.

1.8.6 Replicated data with mutual exclusion

The usual reason for implementing tokens is to obtain mutual exclusion on a shared resource or a replicated data item. In an unconstrained setting, like

the Linda tuple space, we saw that correct behavior may require the use of a synchronous broadcast. What if updates are only done by a process that holds mutual exclusion on the object being updated, in the form of a token for it?

If W_i^k denotes the *k'th* replicated write done by the *i'th* process to hold the token, we will always have $R_i \rightarrow W_i^0 \rightarrow \cdots W_i^n \rightarrow P_i$. Now, since P_i denotes the passing of the token to the process that will next obtain it, it follows that if a process holds a token, then all *write* operations done by prior holders precede the *pass* operation by which the token was obtained. Thus, if CBCAST is used to transmit *write* operations, any process holding the token will also see the most current values of all data guarded by the token.

It follows that we can obtain 1-copy behavior for a replicated variable using a token-passing and updating scheme implemented entirely with asynchronous one-phase broadcasts. Any process holding the token will "know" it also possesses an up-to-date state (this kind of knowledge is formalized in [Taylor88]). Moreover, execution can be done without any delays at all, reading and writing the local copies of replicated variables without delay – just as for a non-replicated variable – and leaving the corresponding broadcasts to complete in the background.

Figure 1.8 illustrates replicated update using token passing in this manner. All the updates occur along the dark lines that highlight the path along which the token travels, which is exactly the $\rightarrow$ relation used in the above argument. Although the system has the freedom to delay updates or deliver them in batches, it can never deliver them out of order or pass a token to a process that has not yet received some pending updates. The algorithm is thus executed as if updates occurred instantaneously.

What about an application that uses multiple data items, and multiple locks? The algorithm described above can yield very complex executions in such a setting, because of delayed delivery of update messages and deliveries that can occur in different orders at different sites. Nonetheless, such a system always has at least one synchronous global execution that would have yielded the same outcome. To see this, observe that $\rightarrow$ for this system is a set of paths like the one seen in Figure 1.8, each consisting of a sequence of *write* and *pass* operations. These paths cross when a token is passed to some process *p* that subsequently receives a second token (Figure 1.9). Such a situation introduces edges that relate operations in the former path to operations in the latter. Similarly, if a process reads a data item, all the subsequent actions it takes will be ordered after all the previous updates

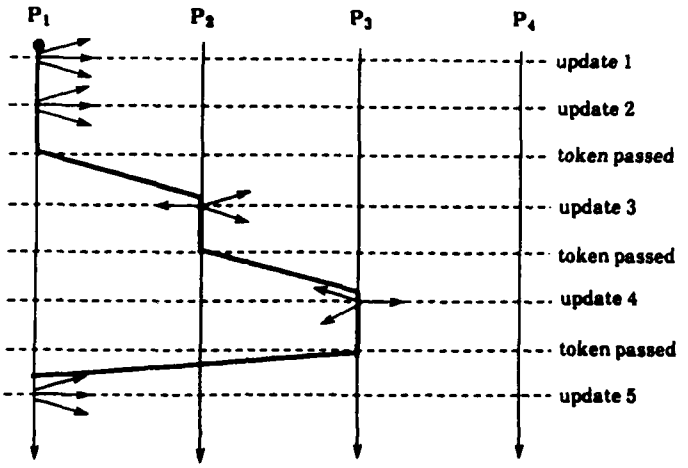

Figure 1.8: A virtually synchronous Replicated update algorithm

to that data item. Although it may be hard to visualize, the resulting $\rightarrow$ relation is an acyclic partial order. It can therefore be extended into at least one total order, and in general many such orders, each of which describes a synchronous global execution that would yield the same values in all the variables as what the processors actually saw.

Thus, although the update algorithm is completely asynchronous and no process ever delays except while waiting for a token request to be granted, the execution is indistinguishable from a completely synchronous one such as would result from using a quorum write [Herlihy86b] for each update. The performance of our algorithm is much better than that of a synchronous one, because a synchronous update involves sending messages and then waiting for responses, whereas an asynchronous update sends messages without stopping to wait for replies. No process is ever delayed in the execution illustrated by the figure, except when waiting for a token to be passed to it.

A similar analysis can be undertaken for replicated data with local read- and replicated write-locks, although we will not do this here. The existence of local read-locks implies that write-locks must be acquired synchronously, with each process granting the lock based on its local state, and the write-lock considered to be held only when all processes have granted it. This leads to an algorithm in which read-locks are acquired locally, write-locks are acquired using a synchronous group broadcast, and updates and lock release are done using asynchronous broadcasts. In a refinement, the breaking of

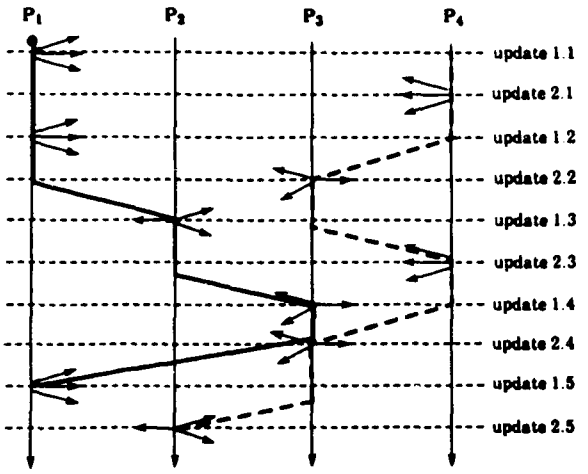

Figure 1.9: Replicated updates with multiple tokens

read-locks after failures can be prevented by asynchronously broadcasting information about pending read-locks in such a way that any updates that depend on a read lock are related to the read-lock broadcast under $\rightarrow$. This method was first proposed by Joseph [Joseph86].

1.8.7 Dealing with failures

The analysis of the preceding section overlooks failures and other dynamic group membership changes. In many applications one wishes to deal with such events explicitly, for example by granting the token to the next pending request in the event that the current holder fails. Recall that group membership changes must be totally ordered with respect to other events in order to ensure the virtual atomicity of broadcast delivery. Since ISIS does this, any broadcast sent in the token or update algorithm will be received by all members of the group that stay operational, and in the same view of the current membership.

In order to pass the token after a failure, we want to implement a rule like this:

> *All group members monitor the holder of the token. If the holder fails, the oldest process in the group takes over and passes the token on its behalf.*

For this simple rule to work, we need to know two things about the system.

First, the rule depends on the ordering of group members by age, which must be consistent from member to member. Group views have this property in ISIS.

Secondly, we need to know that any view change reporting a failure will be ordered after all broadcasts done by the failed process. This ensures that if the failed process did a pass before dying, then either no process saw it happen or all processes saw the broadcast and are already watching the new holder. That is, if X_p^i is the *i'th* action taken by p and F_p denotes the event reporting the failure of p, we need $\forall p, i : X_p^i \rightarrow F_p$. Certainly, in any synchronous execution a failed process takes no further actions, hence this condition will also hold for any virtually synchronous execution.

Thus, one could readily implement a fault-tolerant token-passing algorithm in a virtually synchronous environment.

Notice that the failure ordering property links the atomicity of one broadcast to the atomicity of a subsequent one. A conventional atomic broadcast places an all or nothing requirement on broadcast delivery. But, this does not rule out the transmission of a broadcast a that will not be delivered anywhere because of a failure, followed by the transmission of a broadcast b from the same sender that will be delivered. In a virtually synchronous system, such behavior is not permitted.

1.9 Other virtually synchronous tools

We have seen virtually synchronous solutions to two problems in the list of tools enumerated at the start of this chapter: replicated data management and synchronization. Let us briefly address the other problems in the list.

1.9.1 Distributed execution

There are several ways to distribute an execution over a set of sites in a virtually synchronous setting. The ISIS toolkit supports all of the following:

Pool of servers: The Linda S/Net illustrates a style of distributed execution that we call the *pool of servers*. In this approach, a pool of servers share a collection of work-description messages, extracting them one at a time, performing the indicated operation, and then placing a completion message back into the pool for removal by the process that initiated the work. The approach is simple and lends itself to environments where the processes composing a service are loosely coupled and largely independent of one another. It can be made fault-tolerant by maintaining some sort of

"work in progress" trace that can be located when a process is observed to fail. On the other hand, this method of distributed computing is potentially costly because it relies so heavily on synchronous operations. Were a system to make heavy use of the tuple space, it might become a bottleneck.

Redundant computation: A redundant computation is one in which a set of processes perform identical operations on identical data. The approach was first proposed by Cooper for use in the Circus system [Cooper85]. Redundant computation has the advantage of fault-tolerance, and when the operation involves updates to a replicated state it is often the most efficient way to obtain a replicated 1-copy behavior. On the other hand, it is unclear why one would want to use a redundant computation for an operation that does not change the state of of the processes involved. With the exception of a realtime system operating under stringent deadlines, where it might increase the probability of meeting the deadline, such an approach would represent an inefficient use of computational resources.

Redundant computation is easily implemented in a virtually synchronous environment. The event initiating the computation is broadcast to all the processes that will participate in the computation. They all perform the computation in parallel and respond to the caller, sending identical results. The caller can either continue computing as soon as the first result is received, or wait to collect replies from all participants.

ISIS does not permit redundant computations to be nested unless the application makes provisions to handle this possibility. In contrast, the Circus system supports nested redundant computations in a way that is transparent to the user, even permitting replicated callers to invoke non-idempotent operations and operations implemented by a group with a replication factor different from that of the caller. Cooper discusses these problems, as well as mechanisms for guarding against incorrect replies being sent by a faulty group member, in [Cooper85].

Coordinator-cohort computation: A coordinator-cohort computation is one in which a single process executes a request while other processes back it up, stepping in to take over and complete the request if a failure occurs before the response is sent [Birman87b]. Such a computation could make good use of the parallelism inherent in a group of processes, provided that different coordinators are picked for different requests (in this way sharing the load). However, if the distributed state of the processes involved is changed by a request, the coordinator must distribute the updates made to its cohorts at the end of the computation. The cost of this style of updating would have to be weighed against that of a redundant computation in order

to pick the computational strategy most appropriate for a given application.

Implementation of a coordinator-cohort computation is easy in a virtually synchronous setting. The request is broadcast to the group that will perform the computation. The caller then waits for a single response. In many applications, the broadcast can be done using a one-phase protocol such as CBCAST, although this decision requires an analysis similar to the one used for the token passing example. The participants take the following actions in parallel. First, they rank themselves using such information as the source of the request, the current membership of the group doing the request, and the length of time that each member has belonged to the group. Since all see the same values for all of these system attributes, they all reach consistent decisions. The coordinator starts computing while the cohorts begin to monitor the membership of the process group. The coordinator may disseminate information to the cohorts while doing this, or use mechanisms like the token for synchronization. When the coordinator finishes, it uses CBCAST to atomically reply to the caller and (in the same broadcast) send a termination message to the cohorts. If a coordinator fails before finishing, its cohorts react as soon as they observe the failure event (recall that a broadcast sent prior to the failure is delivered before the failure notification). The cohorts recompute their ranking, arriving at a new coordinator that terminates the operation. If the original coordinator sent information while computing, there are a number of options: the cohorts can spool this and discard it if a failure occurs, or could apply it to their states and take over from the coordinator by picking up from where it died.

Unless the application is sensitive to event orderings, this algorithm can be implemented with asynchronous CBCAST's. As in the case of the token algorithm, a highly concurrent execution would result.

Subdivided computation: A subdivided computation arises when each participant does part of the requested task. The caller collects and assembles these to obtain a complete result. For example, each member of a process group might search a portion of a database for items satisfying a query, with the result being formed by merging the partial results from each subquery. As in the case of a coordinator-cohort computation, the participants in a subdivided computation can draw on a number of properties of the environment to divide the computation. Provided that all use the same decision rule, they will reach the same decision. Dealing with failures, however, is problematic in this case. A simple solution is to identify the results as, e.g., "part 1 of 3." A caller that receives too few replies because some processes have failed can retry the query, or perhaps just the missing

part.

1.9.2 System configuration and reconfiguration

We have treated the system configuration as a synonym for the view of processes groups and processors in the system – that is, a list of the operational members, ordered by age. However, some systems have a software configuration that augments this view-based configuration and is also used for deciding how requests should be processed. This suggests that software designers need access to a broadcast primitive like the one ISIS uses to inform process group members of group membership change. The GBCAST primitive can be used for this purpose. Because GBCAST is atomic and totally ordered with respect to both CBCAST and ABCAST, one can use it to transmit updates to a replicated configuration data structure shared by the members of a process group. Such an update would otherwise be implemented just like any other update to replicated data, but because of the strong ordering property of the GBCAST, all processes see them in the same order with respect to the arrival of other messages of all kinds. Thus, when a request arrives or some other event is observed, the extended configuration can be used as part of the algorithm for deciding how to respond.

1.9.3 Recovery

When a process recovers, it faces a complex problem, which in ISIS is solved by the *process-group join tool*. A recovering process starts by attempting to rejoin any process groups which the application maintains. When invoked, the tool checks to see if the specified process group already exists and if any other process is trying to recover simultaneously. A given process will observe one of the following cases:

1. The group never existed before and this process is the first one to join it. The group is created and the caller's initialization procedure invoked. If two processes restart simultaneously, ISIS forces one to wait while the other recovers.

2. The group already exists. After checking permissions, the system adds the joining process to the group as a new member, transferring the state of some operational member as of *just before* the join took place. The transfer is done by repeatedly calling user-provided routines that encode the state into messages and then delivering these

to user-provided routines that decode the messages in the joining process. The entire operation is a single virtually synchronous event. All the group members see the same set of events up to the instant of the join, and this is the state that they transfer. After the transfer, all the members of the group (including the new member) see the membership change to include the new member, and subsequently all see exactly the same sequence of incoming requests (subject to the ordering constraints of the protocol used to send those requests).

3. The group previously existed but experienced a total failure. The handling of this case depends on whether or not the group is maintaining non-volatile logs and, if so, whether or not this process was one of the last to fail and consequently has an accurate log (Skeen gives an algorithm for deciding this in [Skeen85]). The former case is treated like case (1). In the latter, a recovery is initiated out of the log file. If the process is not one of the last to fail, the system delays the recovery until one of the last group members to fail has recovered, and then initiates a state transfer as in case (2).

   In ISIS, a checkpoint is done by performing a state transfer into a log file. Thus, recovery out of a log looks like a state transfer from a previously operational member, followed by the replay of messages that were received subsequent to the checkpoint and prior to the failure. Management and recovery from logs in a virtually synchronous setting has been examined by Kane [Kane88].

Interestingly, ISIS obtains this powerful mechanism by composing several of the *tools* described earlier. For example, the state transfer is actually done by invoking tools like the coordinator cohort tool described above. To ensure that this is done at a virtual instant in time, the recovery tool uses a GBCAST protocol to add the new member to the existing process group, and triggers the state transfer just before reporting the membership change to the group members (including the new member). The mechanism is not trivial to implement, but is still fairly simple. Similarly, solutions to the other aspects of the problem are constructed out of broadcast protocols and reasoning such as what we described above for the token passing algorithm.

1.10 Orthogonality issues

It was observed that for a set of tools to be of practical value they must permit a step-by-step style of programming. For example, if we build a distributed program using some set of tools, and then extend it in a way that requires an additional replicated variable, the only code needed should be for managing and synchronizing access to the new variable. It should not be necessary to reexamine all the previous code to ensure that no unexpected interaction will creep in and break some preexisting algorithm. We say that a set of tools are orthogonal to one another if they satisfy this property.

A desirable characteristic of the virtually synchronous environment is that orthogonality is immediate in algorithms that require just a single broadcast event, because these broadcasts are virtually synchronous with respect to other events in the system. For example, since updates to a replicated variable appear to be synchronous, introducing a coordinator-cohort computation for some other purpose in a program that uses such updates should not "break" the replicated data mechanism. More complex mechanisms, such as the ISIS recovery mechanism, are made to look like a single synchronous event, even when they involve several distinct subevents. A consequence is that one can build software in ISIS by starting with a non-distributed program that accepts an RPC-style of interaction, then extending it into a distributed solution that uses a process group and replicated data, introducing a dynamically changing distributed configuration mechanism, arranging for automated recovery from failure, and so forth. Each change is virtually synchronous with respect to the prior code, hence no change will break the pre-existing code. The same advantage applies in a setting like the Linda system: software here can be developed by debugging a single process that uses the tuple-space primitives, and once it is operational replicating this process to the extent desired.

1.11 Scaling, synchrony and virtual synchrony

We observed that a genuinely synchronous approach to distributed computing will have scaling problems. In Lindathe broadcast bus is a bottleneck, particularly if load rises to the point where processors begin to experience input buffer overruns. Moreover, as Linda grows, all processors must devote a higher and higher percentage of their CPU cycles to just maintaining the shared memory. A large Linda S/Net system would suffer serious perfor-

mance problems.

Similarly, the performance of the HAS system degrades as a function of the number of sites in the network, because a larger network has larger expected delays over its communication links. This increases the minimum delay before a broadcast can be delivered, and because HAS does not support an asynchronous broadcast the performance of application software is directly impacted.

A system like ISIS has a slightly different problem. Here, the basic protocols are essentially linear in the size of process groups (see [Birman88]). However, several parts of ISIS involve algorithms that scale with the number of sites in the network. To address this issue, the next version of ISIS will introduce a notion of scope into the system. The idea is to bound these algorithms to small collections of sites in a way that does not compromise the correctness of the overall system. It is believed that the resulting system will scale up to several hundred sites without imposing a severe load on any machine, assuming that process *groups* do not grow to include more than 20 or 30 members (here, we assume a 10Mbit network and 2-5MIP workstations). These figures are based on current experience with those aspects of the ISIS system that will not change when better algorithms are installed. Thus, ISIS potentially scales to moderately large networks, but is unlikely to scale up into geographically distributed settings with tens of thousands or millions of sites. An open question is whether there exists some other architecture that would yield virtual synchrony and high levels of concurrency, as does ISIS, but would scale without limit.

Finally, consider the quorum schemes, which also achieve virtual synchrony. These degrade in a way that is completely determined by the quorum size and the number of failures to be tolerated. While process groups stay small, one would expect bounded performance limited by RPC bandwidths, and poorer than what can be achieved using asynchronous protocols. On the other hand, such an approach is unlikely to scale to very large groups.

To summarize, there seems to be good reason to view virtual synchrony as an effective programming tool for small and medium size networks, perhaps even encompassing a typical medium-size factory or company. In much larger settings, other approaches yielding weaker correctness guarantees is needed.

It should also be noted that our collection of tools focuses on programming "in the small". The design and implementation of software for a factory requires something more: a methodology for composing larger systems out of smaller components, and perhaps a collection of tools for programming in

the large. The former would consist of a formalism for describing the behavior of system components (which could themselves be substantial distributed systems) and how components interact with one another, independently of implementation. The latter would include software for cooperative application development, monitoring dependencies between components of a large system and triggering appropriate action when a change is made, file systems with built in replication, and mechanisms with which the network can be asked to monitor for arbitrary user-specified events and to trigger user-specified actions when those events occur. These are all hard problems, and any treatment of them is beyond the scope of this discussion. Moreover, the current state of the art in these areas is painfully deficient. Substantial progress is needed before it becomes practical to talk about building effective and robust network solutions to large-scale problems.

1.12 An example of ISIS software and performance

It might be interesting to see a sample of a typical ISIS program. The program shown below solves the drilling problem in ISIS. In contrast to the Linda S/Net solution, the method is fault-tolerant and supports dynamic process recovery. As before, the code will be in two parts: the code for a process that issues the original work request to the cell controller, and the distributed algorithm run in parallel by the control processes. We start with the code for making a request:

```
/* Define a type called hole_t for describing holes */
typedef struct
{
    /* Description of hole */
    int     h_x, h_y, ....;         /* Description of the hole */

    /* Runtime variables set by algorithm */
    address h_drill;                /* Process that will drill it */
    int     h_state;                /* Status, see below */
} hole_t;

#define     H_NULL      0           /* Initial state */
#define     H_ASSIGNED  1           /* h_drill has been set */
```

```
#define     H_DRILLING  2          /* Drilling underway */
#define     H_DONE      3          /* Hole completed */

main()
  {
    address driller;
    int nholes, nreplies, checklist[MAXHOLES], ntocheck;
    hole_t holes[MAXHOLES];

    ... initialize nholes and holes[0..nholes-1] ...

    /* Lookup address of drill service */
    driller = pg_lookup("/bldg14/cell22-a/driller");

    nreplies = cbcast(driller, WORK_REQ,
                      /* Message to broadcast */
                      "{%d,%d,...,%a,%d}□", holes, nholes,
                   1  /* One reply wanted */,
                      /* Reply format */
                      "{%d}□", checklist, &ntocheck);

    if(nreplies != 1)
        panic("Drill service is not available\n");
    if(ntocheck != 0)
    {
        printf("Job requires manual recheck.  Please check:\n");
        for(i = 0; i < ntocheck; i++)
        {
            hole_t *h = &holes[checklist[i]];
            printf("Hole at %d,%d ...\n", ...);
        }
        printf("Type <cr> when finished rechecking: ");
        while(getchar() != '\n') continue;
    }
    ... etc ...
  }
```

This program imports the list of entry points from the drill service, which defines the **WORK_REQ** entry to which the work request is being transmitted.

To a reader familiar with the C programming language, the code will be self-explanatory except for the arguments to **cbcast**, which are the group to transmit to (a long form accepting a list of groups is also supported), the entry point to invoke in the destination processes, the format of the data to transmit (here, an array of structure elements), the array itself and its length, the number of replies desired (1), the format of the expected reply (an array of integers), a place to copy the reply, and a variable that will be set to the length of the reply array.

The cell controller is more complex:

```
/* A typical drill controller */

# include "hole-desc.h"

main()
  {
    /* Bind the two entry points to handler routines */
    isis_entry(WORK_REQ, work_req);
    isis_entry(DRILLING, drilling);
    /* Start ISIS lightweight task subsystem */
    isis_mainloop(restart_task);
  }

/* Task to restart this process group member */
restart_task()
  {
    /* Join or create group, obtain current state */
    driller = pg_join("/bldg14/cell22-a/driller",
        /* On first time create, call first_time_init */
            PG_INIT, first_time_init,
        /* On joining an existing system, do state transfer */
            PG_XFER, state_xfer_out, state_xfer_in,
        /* Call monitor_routine on membership changes */
            PG_MONITOR, monitor_routine,
        0);
   }

/* Global variables */
int     checklist[MAXHOLES], ntocheck;
```

```
/* Reception of a new work request (WORK_REQ entry) */
work_req(msg)
  message *msg;
  {
    int nholes;
    hole_t holes[MAXHOLES];
    pgroup_view *pgv = pg_getview(driller);

    msg_get(msg, "{%d,%d,...,%a,%d}□", holes, &nholes);
    for(n = 0; n < nholes; n++)
    {
        hole_t *h = &holes[n];
        h->h_who = schedule(h, pgv);
        if(<first hole assigned to this process>)
            h->h_state = H_DRILLING;
        else
            h->h_state = H_ASSIGNED;
    }
    t_fork(drill_task);
    ntodrill = nholes;
    ntocheck = 0;
    cur_req = msg;
    send_rep();
  }

send_rep()
  {
    t_wait(&work_done);
    if(pg_rank(my_address, driller) == 1)
          /* Oldest process replies for group */
          reply(cur_req, "{%d}□", checklist, ntocheck);
    cur_req = (message*)0;
  }

int drill_task_active;

/* How many failures we can tolerate at a time */
#define   N_FAULTS_TOLERATED    1
```

```
drill_task()
  {
    int done_with, n;
    char answ[N_FAULTS_TOLERATED];
    ++drill_task_active;
    n = next_hole(my_address, holes);
    while(n != -1)
    {
        hole_t *h = &holes[n];
        drill_hole(h);
        done_with = n;
        n = next_hole(holes);
        /* Async. broadcast to inform others of my next action */
        cbcast(driller, DRILLING,
                  "%a,%d,%d", my_address, n, done_with,
              N_FAULTS_TOLERATED+1,
          "%c", &answ);
    }
    --drill_task_active;
  }

/* Invoked when a DRILLING cbcast is done */
drilling(msg)
  {
    msg_get(msg, "%a,%d,%d", who, &next, &done);

    /* Update status of holes list */
    holes[done].h_state = H_DONE;
    if(next != -1)
        holes[next].h_state = H_DRILLING;

    /* When done, awaken send_rep() */
    if(--ntodrill == 0)
        t_signal(&work_done);

    /* Confirm that we got the message */
    reply(msg, "%c", '+');
  }
```

```
/* When a process fails, reassign its remaining work */
monitor_routine(pgv)
  pgroup_view *pgv;
  {
       int must_drill = 0;
       if(pgv->pgv_event != PGV_DIED)
           return;
       for(h = holes; h < &holes[nholes]; h++)
           if(h->h_who == pgv->pgv_died)
           {
                if(h->h_state == H_ASSIGNED)
                {
                    h->h_who = schedule(h, pgv);
                    if(addr_ismine(h->h_who))
                        ++must_drill;
                }
                else if(h->h_state == H_DRILLING)
                {
                    h->h_state = H_DONE;
                    checklist[ntocheck]++ = h-holes;
                    if(--ntodrill == 0)
                        t_signal(&work_done);
                }
           }
       if(must_drill && drill_task_active == 0)
            t_fork(drill_task);
  }
```

The above code is certainly longer than for the Linda example, and it may look more complex. However, the Linda example was not fault-tolerant and did not address the scheduling aspects of the problem. Moreover, our solution is actually quite simple. It works as follows.

Each controller process joins a driller process group. The group as a whole receives each request by accepting a message to the `work_req` entry point. In parallel, all members schedule the work, noting which hole each of the other processes is currently drilling and marking all others as assigned. A lightweight task is forked into the background to do the actual drilling; it will share the address space cell controller with the task running `work_req`,

using a non-preemptive "monitor" style of mutual exclusion under which only one task is executing at a time, and context switching occurs only when a task pauses to wait for something. The `work_req` task now waits for drilling to be completed.

The `drill_task` operates by drilling the next assigned hole, then broadcasting to all group members when it finishes this hole and moves on to the next one. The broadcast must be done synchronously, waiting until enough replies are received to be sure that the message has reached at least `N_FAULTS_TOLERATED` remote destinations (because the sender will receive and reply to its own message, we actually wait for one more reply above this threshold). The point here is to be sure that even if `N_FAULTS_TOLERATED` drill processes crash, the broadcast will still be completed because some operational process will have received it. Each group member marks the previous hole as `H_DONE` and the next one as `H_DRILLING` when this broadcast arrives.

If a process fails, the other group members detect this when their monitor routines are invoked by ISIS. They reassign work, moving any hole that the failed process was actually drilling to the check list. Any process that has ceased drilling (and hence no longer has an active `drill_task` spawns a new one at this time.

A normal ISIS application would also include code for initializing the group at cold-start time and for transferring the state of the group to a joining member, by encoding it into one or more messages. We have omitted this code above.

What about performance? Figures 1.10-a and 1.10-b graph the performance of this application program, in holes-per-second drilled by the entire group as a function of the number of members. We generated these figures on a network of SUN 3/60 workstations, otherwise idle, running release 3.5 of the SUN UNIX system and communicating over a 10Mbit ethernet. Figure 1.10-a was based on a control program for which the simulated delay associated with moving the drill units and drilling holes was 1-second per hole. Figure 1.10-b used a delay of zero. In the absence of any ISIS overhead, the first graph would show a linear speedup and numbers would all be infinite in the second graph. Thus, the communication overhead imposed by this version of ISIS becomes significant when the group reaches six members, limiting the attainable speedup for drilling holes with this delay factor. Since the number of messages sent per second grows as the square of the size of the group in this example, these curves are not unreasonable ones. More detailed performance figures for ISIS are available in [Birman87b,Birman88].

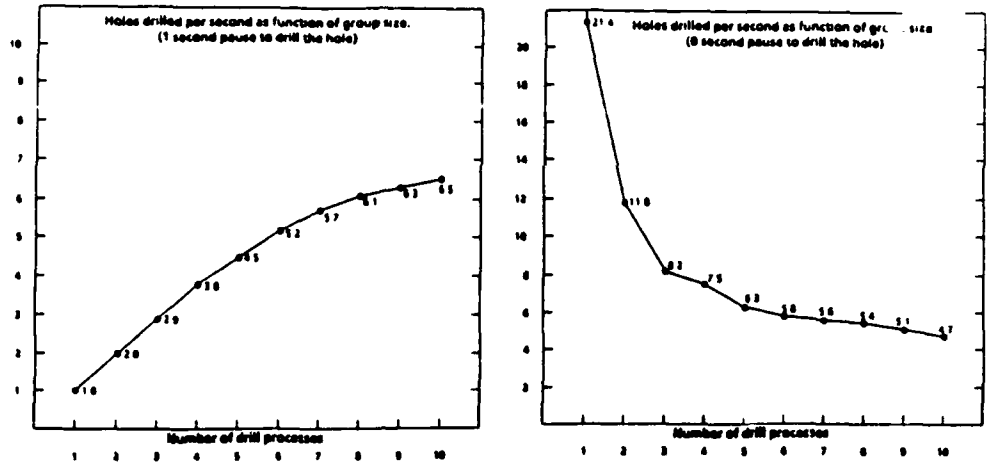

Figure 1.10: Holes drilled per second for different per-hole delays

1.13 Theoretical properties of virtually synchronous systems

We conclude the chapter with a review of some theoretical results relevant to the behavior of virtually synchronous systems.

1.13.1 How faithful can a virtually synchronous execution be to the physical one?

A system like ISIS seeks to provide the illusion of a synchronous execution while actually executing asynchronously. Moreover, unlike Linda or HAS, failures are "events" in the virtually synchronous execution model used by ISIS. This leads to limits on the extent to which the model can be faithful to reality. For example, it is impossible to ensure that a virtually synchronous execution will present failures in the precise sequence that physically occurred with respect to other events. Specifically, in a situation where the system is about to deliver a broadcast, it cannot prevent a physical failure from occurring just as the broadcast delivery is taking place. At one destination, the failure has occurred "after" delivery, but at the other it is "before" delivery. From this we see that a system like ISIS might sometimes be forced

to claim that a message was delivered to a process that had actually crashed before delivery took place.

A relevant theoretical result [Fisher85] shows that it is impossible to reach distributed agreement in an asynchronous system subject to failures. Additional work along these lines was done by Hadzilacos and reported in [Hadzilacos84,Perry86]. These results limit what is achievable in a virtually synchronous system. In particular, this establishes that ISIS cannot avoid all risk of incorrectly considering an operational site to have failed.

On the other hand, it is possible for a system to avoid claiming anything inconsistent with the *observable* world. This is done by introducing agreement protocols to decide what picture of a fundamentally uncertain event to provide in its synchronous world model, and then present this to its users in a consistent manner. This is what ISIS does. Unless a failed site or process recovers and can be queried about what it observed just before failing, code that runs in ISIS can never encounter an inconsistency. Moreover, when there is some question of ensuring that all processes have really observed a broadcast or other event, this can be arranged by briefly running the system synchronously – for example, by asking those processes to reply after they have seen the event and waiting for the replies. This is comparable to deferring external actions in a transactional system until the transaction has reached the prepared-to-commit stage.

1.13.2 When can a problem be solved asynchronously?

Schmuck has looked at the question of when a system specified in terms of synchronous broadcasts can be run correctly using asynchronous ones [Schmuck87]. He defines a system to be *asynchronous* if it admits an implementation in which every broadcast can be delivered immediately to its initiating process, with remote copies of the message being delivered sometime later. Failure broadcasts are not considered, although they could be added to the model without changing any of the results.

Define $\mathcal{S}_{async}$ to be the class of all system specifications describing problems that can be implemented in this efficient, asynchronous manner. Schmuck introduces the concept of a *linearization operator*, a function that maps certain partially ordered sets of events to legal histories. In a theorem he shows that for all specifications S:

$$S \in \mathcal{S}_{async} \Leftrightarrow \exists \textit{ a linearization operator for } S.$$

He proves the only-if direction by showing how to construct an implemen-

tation for a specification S, based on its linearization operator, using a communication primitive similar to CBCAST. The other direction is proved by contradiction. The result establishes that Schmuck's implementation method is *complete* for the class $\mathcal{S}_{async}$, i.e. the method yields a correct implementation for all specifications $S \in \mathcal{S}_{async}$.

Schmuck's construction method depends on finding a linearization operator for a given specification. Unfortunately, whether a specification S is in $\mathcal{S}_{async}$ is undecidable. It is immediate that there exists no general method for finding a linearization operator for S. However, Schmuck does propose methods for solving this problem for certain subclasses of $\mathcal{S}_{async}$. The basic characteristic of these subclasses is that they have linearization operators determined entirely by commutativity properties of the broadcasts done in the system. These results can be used to "automatically" construct a linearization operator, and hence an optimal asynchronous broadcast protocol, for a problem like the token-passing ones described above. In fact, when we informally described a way to "linearize" executions on a system with token passing and replicated updates, we essentially described the construction of a linearization operator for that problem. Thus, Schmuck's work formalizes a style of argument that has direct and intuitive meaning.

Herlihy and Wing have also looked at the cost of achieving "locally" ordered behavior in distributed systems. This work develops a theory of linearizability, a property similar to serializability, but observed from the perspective of the objects performing operations rather than from the perspective of the processes acting upon those processes [Herlihy87].

1.13.3 Knowledge in virtually synchronous systems

Some recent work applies logics of knowledge to protocols similar to CBCAST and ABCAST. The former problem was examined by Taylor and Panagaden [Taylor88], who develop a formalism for what they refer to as *concurrent common knowledge*. This kind of knowledge is obtained when an asynchronous CBCAST is performed by a process that subsequently behaves as if all the destinations received the message at the instant it was sent. In ISIS, such a process will never encounter evidence to contradict this assumption. Taylor and Panagaden formally characterize the power of this style of computation, and then use their results to analyze algorithms like the concurrent update discussed above.

Neiger and Toueg have examined the relationship between the total ordering of events in an ABCAST protocol and the total ordering that re-

sults from incorporating a shared realtime clock into a distributed system [Neiger87]. They characterize the settings under which a broadcast algorithm written to use a distributed clock could be implemented using an ABCAST protocol and a logical clock [Lamport78].

1.14 Acknowledgements

We are grateful to Frank Schmuck for detailed comments and suggestions that lead to substantial revisions to this manuscript. The above treatment also benefited from discussions with Ajei Gopal, Ken Kane, Keith Marzullo, Gil Neiger, Pat Stephenson, Kim Taylor, Sam Toueg, Doug Voigt, and many others.

Bibliography

[Bernstein83] Bernstein, P. and Goodman, N. The failure and recovery problem for replicated databases. *Proc. Second ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec (Aug. 1983), 114-122.

[Birman88] Birman, K.P., Joseph, T. and Schmuck, F. Issues of scope and scale in the ISIS distributed toolkit. *In preparation.*

[Birman87a] Birman, K.P. and Joseph, T. Reliable communication in an unreliable environment. *ACM Transactions on Computer Systems.* (Feb. 1987), pp. 47-76.

[Birman87b] Birman, K.P. and Joseph, T. Exploiting virtual synchrony in distributed systems. *Proc. 11th ACM Symposium on Operating Systems Principles.* (Nov. 1987), pp. 123-138.

[Birrell84] Birrell, A. and Nelson,, B. Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems* **2**, 1 (Feb. 1984) 39-59.

[Cheriton85] Cheriton, D. and Zwaenapoel, W. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems* **3**, 2 (May 1985), 77-107.

[Carriero86] N. Carriero and D. Gelertner. he Linda S/Net's Linda Kernel. *ACM TOCS* **4**, 2 (May 1986), 110-129.

[Cristian86] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *IBM Research Report*, RJ 5244 (54244), July 1986.

[Cristian88] Flaviu Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *IBM Research Report*, RJ 5964 (59426), March 1988.

[Cooper85] Cooper, E. Replicated distributed programs. *Proc. 10th ACM Symposium on Operating Systems Principles.* (Nov. 1985), pp. 63-78.

[Fisher85] Fisher, M., Lynch, N., Patterson, M. Impossibility of distributed consensus with one faulty process. *J. ACM* **32**, 2 (Apr. 1985) 274-382.

[Hadzilacos84] Hadzilacos, V. Issues of fault tolerance in concurrent computations. *Ph.D. Dissertation.* Harvard University, June 1984. Available as Technical Report 11-84.

[Herlihy86a] Herlihy, M. Optimistic concurrency control. *Proc. 5th ACM Symposium on Principles of Distributed Computing.* Calgary, Canada (Aug. 1986), 206-217.

[Herlihy86b] Herlihy, M. A quorum-consensus replication method for abstract types. *ACM Transactions on Computer Systems* **4**, 1 (Feb. 1986), 32-53.

[Herlihy87] Herlihy, M. and Wing, J. Linearizable abstract types. *Proc. ACM Symposium on Principles of Programming Languages*, Munich, Germany (Jan. 1987).

[Herlihy88] Herlihy, M. The AVALON Language. *Trying to track down a reference...*

[Joseph86] Joseph, T. and Birman, K. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems* **4**, 1 (Feb. 1986), 54-70.

[Kane88] Kane,. K. Log-based recovery in asynchronous distributed systems. *Ph.D. Dissertation.* Dept. of Computer Science, Cornell University (expected Dec. 1988).

[Kronenberg86] . Kronenberg, N. *et. al.* VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems* **4**, 2 (May. 1986), 130-152.

[Lamport78] Lamport, L. Time, clocks, and the orderings of events in a distributed system. *Commun. ACM* **21**, 7 (July 1978), 558-565.

[Lampson86] Lampson, B. Designing a global name service (1985 Invited Talk). *Proc. 5th ACM Symposium on Principles of Distributed Computing.* Calgary, Canada (Aug. 1986), 1-10.

[Liskov83] Liskov, B. and Scheifler, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems* **5**, 3 (July 1983), 381-404.

[Lynch86] Lynch, N., Blaustein, B., and Siegel, M. Correctness conditions for highly available replicated databases. *Proc. 5th ACM Symposium on Principles of Distributed Computing.* Calgary, Canada (Aug. 1986), 11-28.

[Neiger87] Neiger, G. and Toueg, S. Substituting for realtime and common knowledge in asynchronous distributed systems. *Proc. 6th ACM Symposium on Principles of Distributed Computing.* Vancouver, Canada (Aug. 1987),281-293.

[Perry86] Kenneth J. Perry and Sam Toueg. Distributed Agreement in the Presence of Processor and Communication Faults. *IEEE Transactions on Software Engineering, SE-12*, 3 (Mar. 1986), 477-482.

[Schmuck87] Schmuck, F. The use of efficient broadcast protocols in asynchronous distributed systems. *Ph.D. Dissertation.* Dept. of Computer Science, Cornell University (Aug. 1988).

[Schwartz84] Schwartz, P. and Spector, A. Synchronizing shared abstract types. *ACM Transactions on Computer Systems* **2**, 3 (Aug. 1984), 223-250.

[Skeen85] Skeen, D. Determining the last process to fail. *ACM Transactions on Computer Systems* **3**, 1 (Feb 1985), 15-30.

[Spector88] Spector, A., Pausch, R. and Bruell, R. Camelot: A Flexible, Distributed Transaction Processing System. *Proc. Compcon 88*, San Francisco, CA (Feb 88), 432-437.

[Taylor88] Taylor, K. and Panagaden, P. Concurrent common knowledge: A new definition of agreement for distributed systems. *Proc. 7th ACM Symposium on Principles of Distributed Computing.* Toronto, Canada (Aug. 1988).